Model Driven Software Development

Hype or Panacea

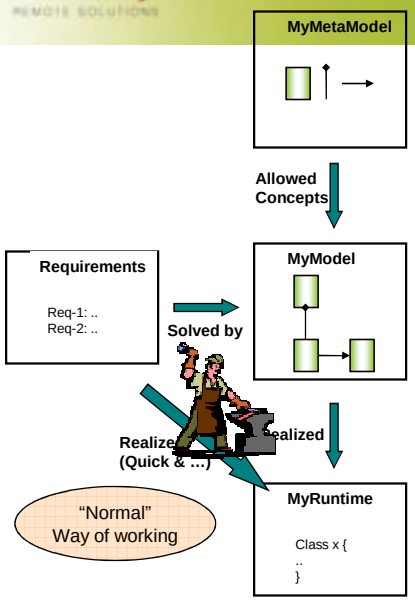
Paul Zenden

16-09-2008

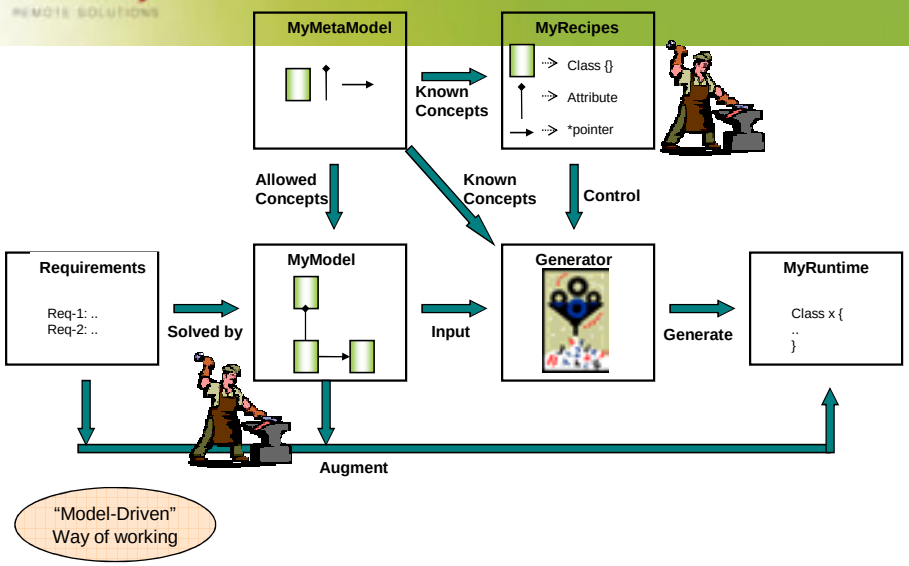
***Panacea*:**
-Solution to all problems
-Universal remedy

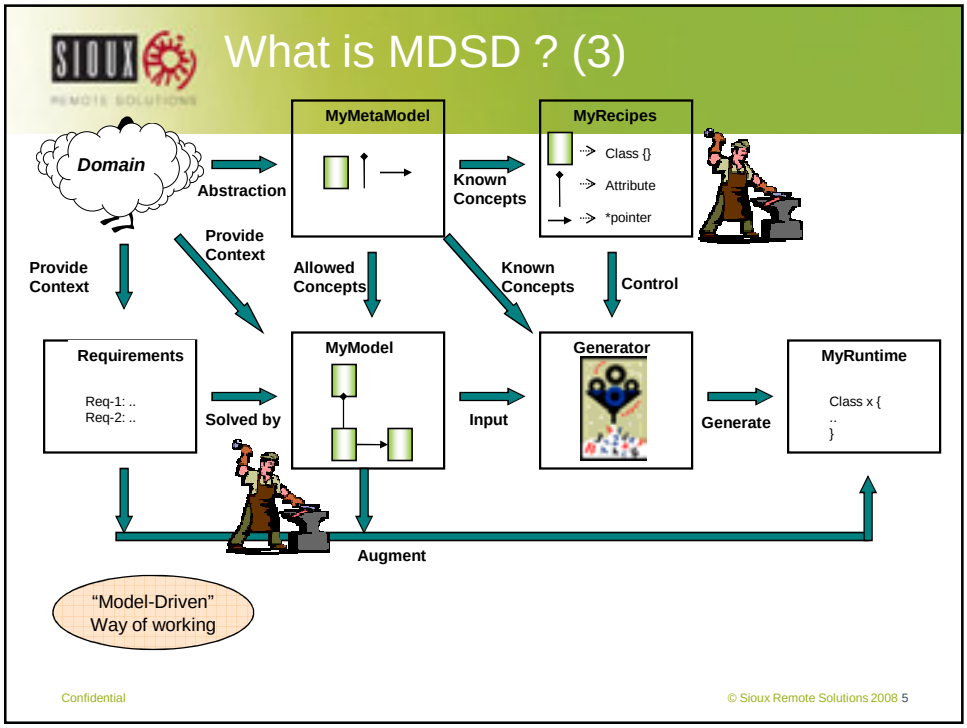
- Let's start with a ***demo***:
  - Get a feeling what I'm going to talk about
  - Case: Domain Persistency Layer
    - From:
      - Domain Model (UML Class model – Enterprise Architect)
    - Generate:
      - Physical database model (SQL create scripts)
      - nHibernate mapping definitions (XML files)
      - Entity Layer (C# Source code)
      - Spring Injection Definitions (XML file snippet)

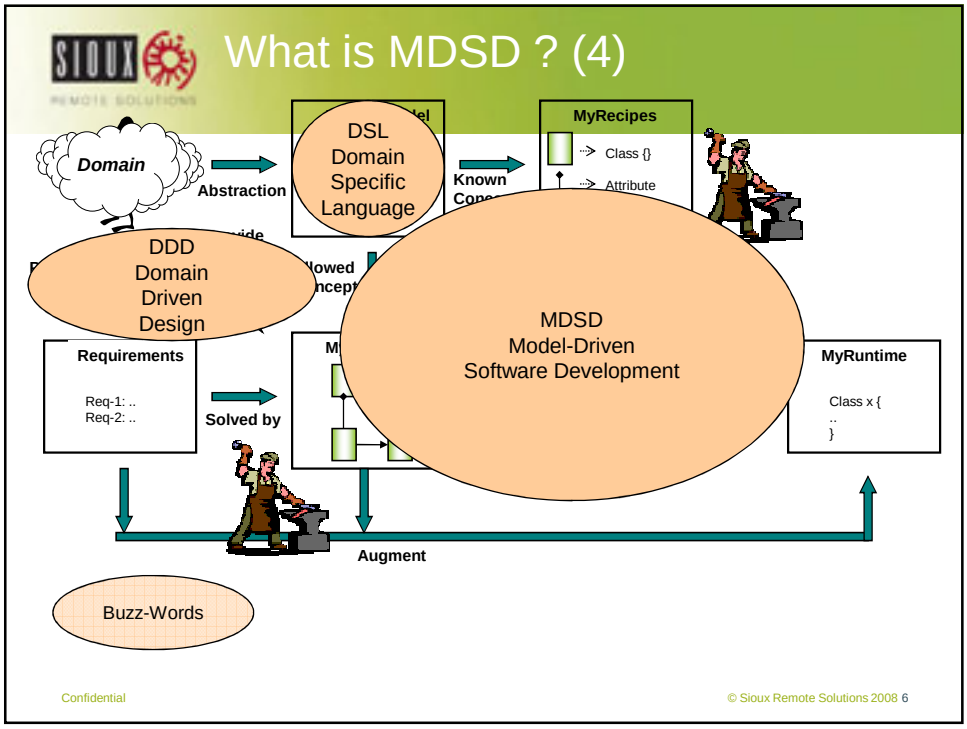
What is MDSD ? (1)



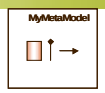
What is MDSD ? (2)




What is MDSD ? (3)
Domain
Abstraction
MyMetaModel
Known Concepts
MyRecipes
Class {}
Attribute
*pointer
Provide Context
Provide Context
Allowed Concepts
Known Concepts
Control
Requirements
Req-1: ..
Req-2: ..
Solved by
MyModel
Input
Generator
Generate
MyRuntime
Class x {
..
}
Augment
"Model-Driven" Way of working
Confidential
© Sioux Remote Solutions 2008 5


What is MDSD ? (4)
Domain
Abstraction
DSL Domain Specific Language
MyRecipes
Class {}
Attribute
DDD Domain Driven Design
MDSD Model-Driven Software Development
Requirements
Req-1: ..
Req-2: ..
Solved by
MyRuntime
Class x {
..
}
Augment
Buzz-Words
Confidential
© Sioux Remote Solutions 2008 6

Meta Model / DSL: Example (1)



Meta Model / DSL

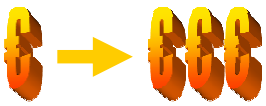
Abstract Syntax
(describes the concepts, their semantics, rules, constraints, etc.)

Concrete Syntax
(describes the notation, visualization, style guidelines)

Confidential

Why we want to invest …

- Day #1 test & integration
- Improved communication developers ⇔ domain experts
- Re-usability within the domain
- Increased innovation power
- Understand before you build
- Right product, right effort
- …



… some more reasons

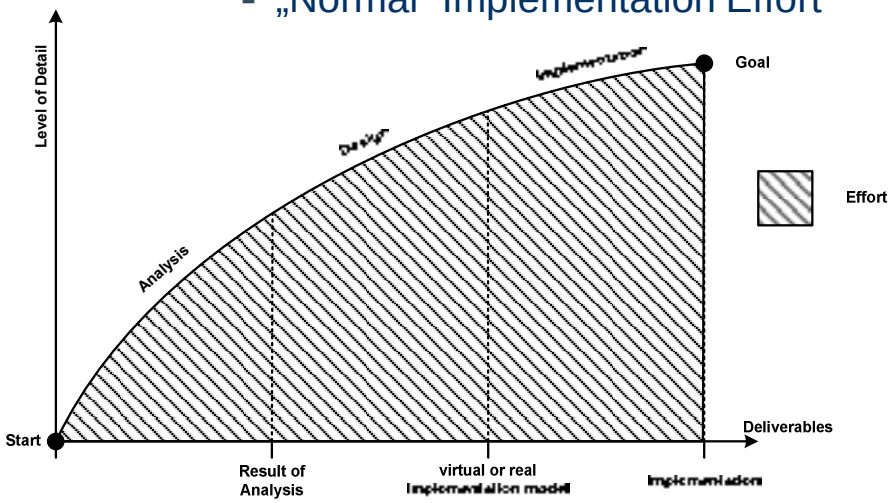
- Economic Benefits
  - Faster implementation of new business requirements.
  - Lower introductory costs of new technologies.
  - Reduced costs during the entire product lifecycle.
- Increase productivity
  - Artifact generation (code, db models, mapping files, etc.)
  - Less unit-testing, more system level testing
  - Increased maintainability
    - Changes through models
    - Decoupling Application and Technology life-cycle
- Increase System Quality
  - Automation reduces potential for errors
  - Improved quality of design
    - Shift focus from implementing to analysis/modeling
  - Early integration
    - Generation stubs of external components based on interface definition

The 'Classic' Approach (1)

- Today's development recipe:
  1. Specify ((e-)paper)
  2. Design ((e-)paper)
  3. Code
- The results
  - Un-maintained documentation
  - Communication abyss between domain expert and developer
  - Continuously reinventing the wheel
  - Wrong product
  - …

Economics (1)

- „Normal“ Implementation Effort



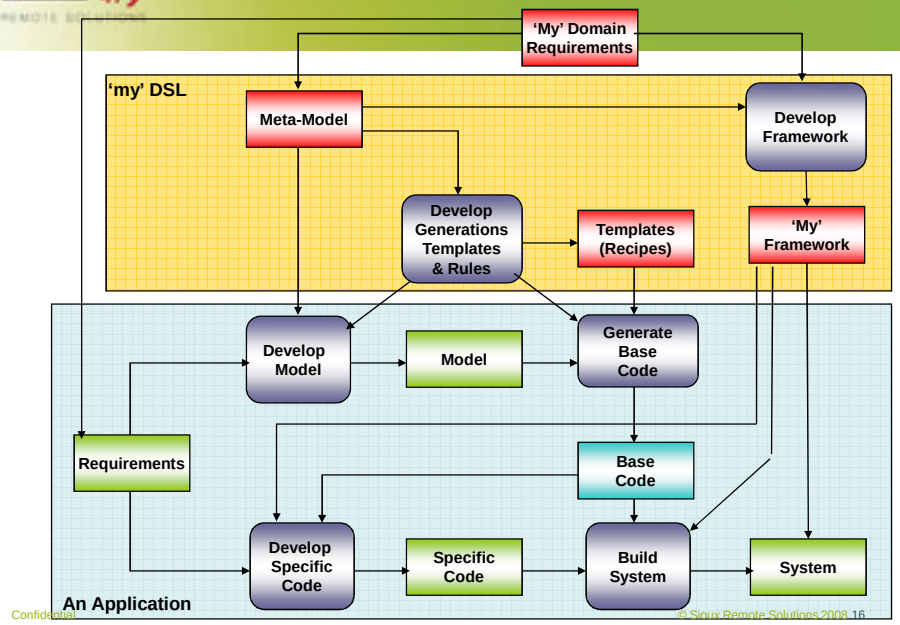
Source: http://www.voelter.de/

Tomorrow’s Approach (1)

- Step into the future
  1. Construct a ***domain specific language*** (DSL)
  2. Model using the DSL
  3. Generate (multiple targets\platforms) code from the Model(s)
- Additional bonuses
  - Integration and test from day #1
  - Facilitates concurrent engineering*)

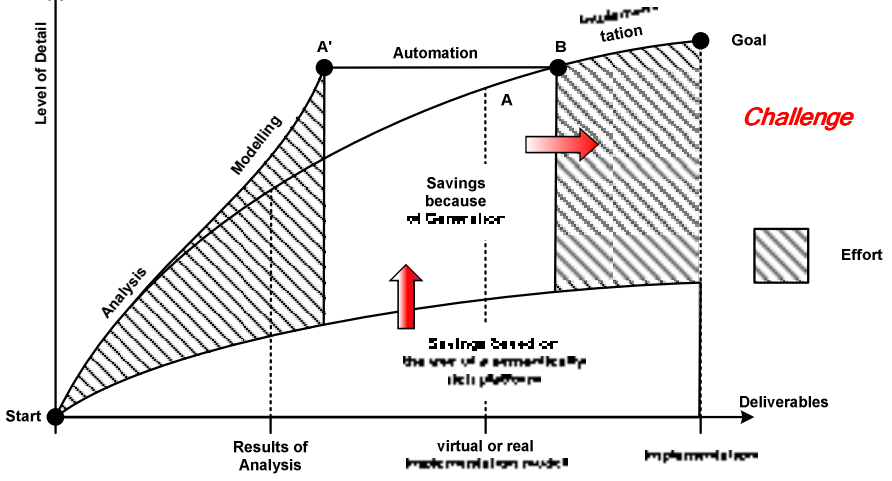
*): Concurrent software engineering but also multi-disciplinary (SW-HW) engineering facilitated, based on:
- HW interfaces can be modeled too: e.g. generate stubs during development
- Model-to-Model transformations allow for retrieving information from algorithm or HW focused models (e.g. Mathlab models)

Tomorrow's Approach (2)



Economics (2)

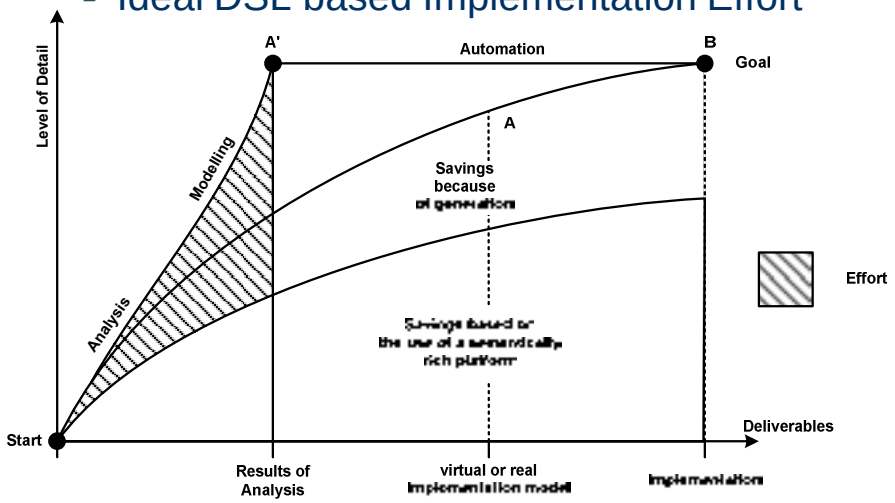
- Realistic DSL based Implementation Effort



Source: http://www.voelter.de/

Economics (3)

- Ideal DSL based Implementation Effort



Source: http://www.voelter.de/

Tool-chains

- Used in 'MDSD-exploration':
  - Eclipse Modeling Project
    - http://www.eclipse.org/modeling/
  - openarchitectureWare
    - http://www.openarchitectureware.org/
  - EnterpriseArchitect
    - http://www.sparxsystems.com/
- Many other tools available
  - Each with their weaknesses and strengths

Demo Case: Revisited (1)

- Goal:
  - Model sub project concepts at domain model level (UML)
    - Use special UML Profile for enforcing meta model concepts
  - Generate small (vertical) piece of complete functionality to implement
    - Business Entity layer (C#)
    - ORM layer (Nhibernate mapping files)
    - SQL create scripts (SQL Server)
    - Not covered by generation:
      - Adaptors, Configuration, Factories, Façades
      - Are, however, possible candidates for future extensions of generator
  - Make clear distinction between generated code and “allow-to-change” code.

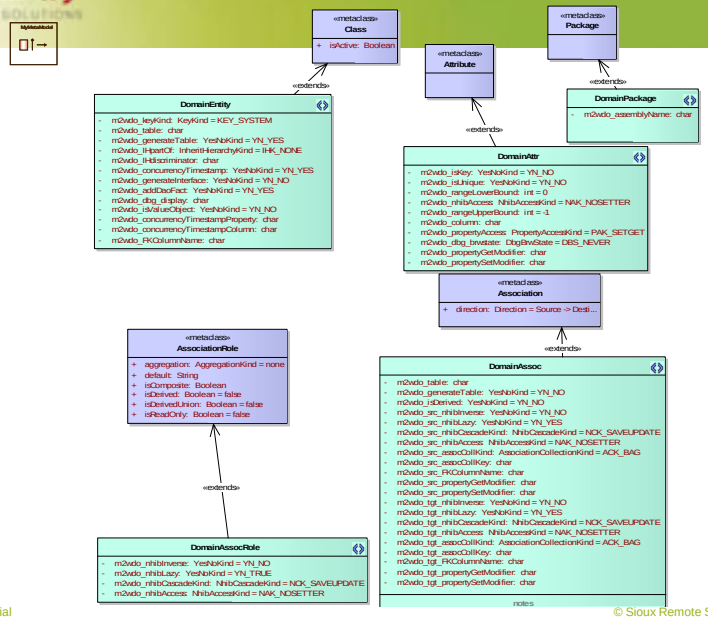
Demo Case: Revisited (2)

- Challenges
  - Meta model concepts growing
  - Reference implementation parallel with generator development
  - Not all model information in EA can be exported to oaW
- Approach
  - 2 teams:
    - Meta model/Generator development
    - Reference implementation/integration and testing intermediate generator outcome
  - Iterative development of generator
  - All issues written down and solved
  - Generate only (mostly) for concepts defined in UML profile

Demo Case: Revisited (3)

- UML Profile
- Domain model
- Templates
- Generated artifacts:
  - C# Code
  - Nhibernate mappings
  - SQL create scripts

Meta-Model (UML Profile)



Model (MyModel)

DomainEntity
DomainAttrib
DomainAssoc

Tagged Values

Confidential

Generator & Generated

Recipes

C#

```
namespace «packageName()» {

/// <summary>
/// «asDomainEntityName()»Base class
«IF this.ownedComment.size > 0» «FOREACH this.ownedComment AS oc»/// «oc.body»
«ENDFOREACH» «ENDIF -»
/// </summary>
[Serializable]
«IF this.m2wdo_dbg_display.length > 0»[DebuggerDisplay(«this.m2wdo_dbg_display»)]«E
public abstract class «asDomainEntityName()»Base
«IF (
(superClass.size>0) ||
(getImplementedInterfaces().size > 0) ||
(this.m2wdo_isValueObject.name == "YN_NO") ||
(this.needsIComparable()) ||
(this.m2wdo_generateInterface.name == "YN_YES")
)» :
«IF (superClass.size>0)»
«IF !isFirstSuperOrInterface(false)»,«ENDIF -» «REM»Variable to prevent , before first interface, so this call
«IF ((superClass.get(0).getAppliedStereotypes().size > 0)  && (superClass.get(0).getAppliedStereotypes().ge
«superClass.get(0).asDomainEntityName() -»
«ELSE -»
«superClass.get(0).name -»
«ENDIF -»
«ENDIF -»
«IF getImplementedInterfaces().size > 0 -»
«IF !isFirstSuperOrInterface(false)»,«ENDIF -»
«FOREACH getImplementedInterfaces() AS ii SEPARATOR ", "» «ii.name»«ENDFOREACH -»
```

Generated Artifacts

C#

```
namespace Sioux.Machine2World.Model20{

/// <summary>
/// MachineEntityBase class
/// </summary>
[Serializable]
[DebuggerDisplay("Id:{id}, Name:{name}, Irc:{irc}, Uri:{uri}")]
public abstract class MachineEntityBase
 :
IMachineEntity
, IBusinessEntity
, IComparable
{
 . . .
```

SQL

```
EXEC MDL_DropTable 'M2W_MACHINES';
GO
CREATE TABLE M2W_MACHINES (
Id uniqueidentifier NOT NULL DEFAULT newid()
,Ts datetimeNOT NULL DEFAULT GETDATE()
,Text5 nvarchar(50)NULL
,Text4 nvarchar(50)NULL
,Text3 nvarchar(50)NULL
,Text2 nvarchar(50)NULL
,Text1 nvarchar(50)NULL
,State intNULL
. . .
```

NHibernate

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" assembly="Sioux.Machine2World.Model20" namespace="Sioux.Machine2World.Model20">
<class name="MachineEntity" table="M2W_MACHINES" lazy="false">
<id name="Id" type="Guid" column="Id" access="nosetter.camelcase">
<generator class="assigned" />
</id>
<timestamp column="Ts" name="LastModified" access="nosetter.camelcase"/>
<property name="Text5" column="Text5" type="System.String" access="nosetter.camelcase"/>
<property name="Text4" column="Text4" type="System.String" access="nosetter.camelcase"/>
<property name="Text3" column="Text3" type="System.String" access="nosetter.camelcase"/>
<property name="LegacyState" column="State" type="System.Int32" access="nosetter.camelcase"/>
<property name="Uri" column="Url" type="System.String" access="nosetter.camelcase"/>
. . .
```

Major Findings (1)

- Getting the reference implementation right is also challenging and takes time
- Tagged values in UML Profile explodes because of regular UML concepts not exported to oaW
- Understanding the generator templates is tough for outsider
- Explicit and implicit rules, settings and dependencies in the UML Profile must be really clear to the implementation team
- Modularizing the templates is essential to keep overview.
- Requires discipline to consequently make changes through the model where the concept is defined in the model
  - E.g.: Comments, Member operations, Realizing interfaces
  - Changing the code is easier than resolving the issue in: the model, or the meta-model, or the generator

Major Findings (2)

- It takes time to get experience with oaW
  - Find out how to find model concepts
  - Available language constructs
  - Smart way of solving small problems in the templates
- Separation of generated code and specific code is difficult to manage.
  - A lot of code tends to move the specific code (generate once), or not generated at all
  - Used generated base classes and specific derived classes – implementation teams felt itself limited in what it can do - are other solutions better?
- 8 iterations before generator output was OK.
  - Duration about 6 weeks
  - Effort:
    - Generator Team : ~150 hours
    - Reference Implementation Team*) : ~ 110 hours

*):
Integrate/test generator output
Define reference implementation
Augment generated code with specifics

Learned from M. Völter (1)

6 sep 2007
Markus Völter
www.voelter.de

- Additional neat features oaW 4.2 (we used 4.1.2)
  - Like debugger for your templates
- Use conditional generating to add tracing in generated source back to generator
- Modularize complex constructs more into separate functions
- Add model constraints checking
  - Don't generate if the model isn't compliant with the constraints.
- For UML drawing:
  - Consider other tool (MagicDraw), or
  - Consider paying developer of importer for adding more functionality
- Make a "How-to" guide:
  - Describe, by example, how to use the concepts of the meta-model
  - Describe how to use the generator
  - Etc.

Learned from M. Völter (2)

- Strongly consider a proprietary meta model to base your generation upon:
  - Only use UML for front-end (drawing of the model)
    - And use Model-2-model transformation to transform into model based on proprietary meta-model.
  - UML Meta-model is complex and huge. Allows many variations which, in principle, :
    - the generator either should support or
    - the constraints checking should check on their absence.
  - Allows future migrations of front-end without changing the generator (only the m2m transformation)
- Consider building 2 models:
  - Domain model in UML
  - Additional implementation related aspects in text
    - As alternative for "tagged values"

Conclusion

- MDSD: Not a Hype but also not a Panacea
- Essential:
  - Clear understanding of thought-process
  - Clear understanding of meta model (e.g. UML) if developing generator recipes
  - Reference implementation required for developing generator recipes
  - Support MyModel builders with “How-to” guides
- Using the tool-chain is easy, however:
  - MyModel contains more detail and must be more precise → Requires more discipline.
  - Need to learn to exploit the tool-chain
- Look out for opportunities:
  - All repeating work is potential opportunity for MDSD
- Many opportunities and a lot of work to be done …

Are you still

Confused or Puzzled

Source of your connected world.

www.sioux.eu
paul.zenden@sioux.eu
+ 31 (0) 40 26 77 100